PRICE $ 2.00

THE 80 NOTEBOOK

SEPTEMBER 1980 ISSUE #6

NOTE: The term "TRS-80" is a registered trademark of Radio Shack, a division of Tandy Corporation. THE 80 NOTEBOOK is not affiliated with Radio Shack or Tandy Corporation in any way.

CYBERMATE ANNOUNCES LOW PRICES AND EXPANDED SERVICES

CYBERMATE has announced some special sale prices for its products and services and an expansion in the scope of its services. Its package of 41 program listings for the TRS-80 Model I Level 2 with 16K covering games, natural language, astronomy, graphics, data base management and word processing has been reduced to $9.00 while individual program listings have been reduced to 90¢ each and individual programs on cassette tape have been reduced to $2.50 each. Orders received from outside the U.S.A. must include $1.50 for postage and handling.

CYBERMATE's monthly "unusual programming" publication for the TRS-80 Model I Level 2, THE 80 NOTEBOOK, has reduced its subscription rates to 95¢ for a sample copy, $11.00 per year in the U.S.A., $16.00 per year in Canada, and $23.00 per year for air mail delivery outside the U.S.A. and Canada.

With 11 years of programming experience, CYBERMATE has expanded its operation to be available for business, scientific, education, operating systems and general programming assignments on a contract basis at a limited time special rate of $11.00 per hour of work required. All programming assignments must be for a TRS-80 Model I Level 2 with the cassette tape, minifloppy disk and sufficient memory recommended for the job at the time a quotation is given. The programming can be done in either BASIC or Assembly language and will include complete system, program and user documentation. All contract programming is guaranteed to meet the specifications requested with program maintenance services available. To order or for further information on any of the above products or services, write to CYBERMATE, 5967 Sullivan Trail, Nazareth, PA 18064 or phone 215-759-6873.

MACHINE LANGUAGE AND ASSEMBLY LANGUAGE: PROGRAMMING MADE AS SIMPLE AS BASIC

In the course of your programming experience with Level 2 BASIC, you will realize the execution speed limitations of using BASIC - particularly in programming graphics. When you reach this level of capability, you may wish to go beyond BASIC into the high speed world of your TRS-80's Z-80 machine language. To do this, you must first understand how your TRS-80 functions internally.

As you may know, the memory in your computer is made up of cells called bytes. Each byte can hold a character, a small number, or graphic display information. Each byte is made up of 8 bits. Each bit can represent a 1 or a Ø depending on whether the bit is on or off magnetically. This string of 8 bits can then form a binary number ranging in value from Ø to 255.

This can best be illustrated by a study of the numeric conversion table found on page G/1 of the Radio Shack Level 2 BASIC Manual. As shown, each bit position containing a 1 can be translated into a decimal value. By summing all 8 potential values in the byte, a numeric value for the byte can be obtained. From left to right, bit 1 is 128, bit 2 is 64, bit 3 is 32, bit 4 is 16, bit 5 is 8, bit 6 is 4, bit 7 is 2, and bit 8 is 1.

These numeric values can be further translated into how the TRS-80 interprets them when a byte is to be treated as a character. These character equivalents can be studied on page C/2 of the manual. For example, the alphabet A thru Z is the numeric range 65 thru 90.

By studying the video display worksheet on page E/1 of the manual, you can see that each character position in the display memory holds 6 graphic positions. A graphic byte can be identified by a 1 in the bit 1 position and bits 3 thru 8 holding the 6 graphic points. Because individual bits can be turned on and off in a byte, several graphic points can be on at one time in a byte. The use of a byte as graphic display information is only valid in the area of memory assigned to the video display.

In viewing the 6 graphic points as arranged in any character position on the worksheet, the upper left point is bit 8, the upper right point is bit 7, the middle left point is bit 6, the middle right point is bit 5, the lower left point is bit 4 and the lower right point is bit 3. In a graphic byte, bit 2 is not used.

Each byte in the computer memory can be located by its address. The first byte is at address Ø, the second byte at address 1 and so on. For example, the video display memory starts at address 1536Ø.

Now you might ask where in memory you can place your machine language programs. The answer to this question depends on how much memory your computer has and how large the BASIC programs, if any, that are going to execute the machine language routines you have loaded into memory are. If a BASIC program is to call one or more machine language routines, you must know exactly how much memory is allocated by your BASIC program versus the total memory in the computer. The remainder of available memory must be sufficient to load and execute the machine language routines. The minimum sixe BASIC program that must be considered will be the relocating loader program which we will discuss later. If you have a 16K machine, the end of memory is at address 32767, a 32K is at 49151, and a 48K is at 65535.

To fix the address of the start of the machine language program area at the tail end of the computer, enter the address of the highest memory location your BASIC program will use (when it is loaded and executed) to the "MEMORY SIZE" prompt at the time of power-up. Since the BASIC program area of memory starts at address 17129, the total memory which your BASIC program can use would be from 17129 to the address entered to the "MEMORY SIZE" prompt. For example, if you have a 16K computer using a BASIC program needing less than 15K

of memory and a machine language routine needing 9ØØ bytes, the reply to the "MEMORY SIZE" prompt should be 31865. This will leave a few bytes between the two program areas as a buffer and starts the machine language area on an even address (something useful in some machine language programs but not necessary).

Now that we have talked a little bit about memory and where a machine language program might be stored for execution, let's take a look at what machine language is and how it is represented in the computer memory. First, machine language is the set of instructions which the electronic circuits in your TRS-80 recognize and carry out when they are found within a program area in the computer memory. When you power up your TRS-80, the circuits start looking for and executing the instructions it finds starting at address Ø. As instructions are carried out, the computer can jump around from program area to program area or within a program area as needed to carry out a series of tasks performed by the programs in memory. BASIC itself is a large program stored to the first 16K of memory. This memory is a special block of memory composed of ROM. The ROM hardware allows the circuits to fetch and execute instructions contained in that section of memory but not to store any new instructions or data in that memory. Because of this, BASIC uses the memory addressed above the ROM to store and manipulate data and execute BASIC program code.

Internally, machine language instructions are composed of an operation code followed by any required operands. Normally, an op code is a number stored in a single byte. The operands which may follow an op code, as required, in successive bytes include an address, a single character, a register I.D., a numeric value or a displacement value. Some operands are combined and stored in a single byte while some op codes take up 2 successive bytes with no operands.

To understand some of these operands, you must understand the architecture of the TRS-80 computer from the internal aspect. Besides the main memory in the computer, which we discussed earlier, the computer has a special, separate block of memory divided into areas of one or two bytes called registers.

The PC register holds the 2 byte address pointing to the location in main memory where the next instruction to be executed by the computer can be found. As instructions are executed, the number of bytes taken up by the instruction and its operands is added to the PC register so that the next instruction's address is always available to the computer. If an instruction requiring a change in address in the PC register is executed, the new address is stored in the register rather than the normal incrementing.

The A register is a 1 byte accumulator used for a variety of functions by many instructions.

The F register is a 1 byte flag containing 8 bit flags used by the computer to show the result of arithmetic and comparison (logical) instructions. When a bit flag is 1, it signals that its representative condition exists.

Flag 1 indicates the sign of the A register. When using the A register as a signed accumulator (by some instructions) so that a range of values from -128 to +127 are possible, the first bit of the accumulator is stored in the sign flag and is treated as a non-numeric bit showing the sign. A sign flag of 1 represents a negative number in bits 2 thru 8 of the accumulator.

Negative numbers are expressed in twos compliment which means that each of bits 2 thru 8 that would be set to 1 for a normal number would be set to Ø and a Ø bit would be set to 1 (bit inversion).

Flag 2 indicates that the result of an arithmetic instruction on the A register is zero or that the result of a comparison was equal.

Flag 3 is not used by the computer but can be set and examined by a program allowing you to use it as a status flag.

Flag 4 is a half carry flag. This will be discussed later in the section covering the instructions that affect it.

Flag 5 is also not used by the computer.

Flag 6 is the parity/overflow flag. This flag is set when the result of an arithmetic instruction on the A register is greater than +127 or less than -128 or when the number of 1 bits in the A register after a logical or bit shift instruction is an even count.

Flag 7 is the add/subtract flag used to indicate what arithmetic instruction was last performed.

Flag 8 is the carry flag set when an 8 bit add instruction generates a carry (or 9 bit result, greater than 255) or an 8 bit subtract instruction requires a borrow (or 9 bit minuend in the A register, greater than 255).

There is a set of general purpose 1 byte registers identified as B, C, D, E, H and L which can also be used by certain instructions as 2 byte register pairs for address manipulation or 16 bit numeric manipulation. These register pairs are identified as BC, DE and HL. It should be noted here that the A and F registers can also be used as a register pair with some instructions. When this is done, the pair is identified as AF. Registers A, F, B, C, D, E, H and L have a set of alternate registers for each main register. This allows you to use 2 A registers, etc. A simple exchange instruction allows you to change which set of registers a following logical or arithmetic instruction will address.

The IX and IY registers are index registers used to hold the address of an area in main memory where data is to be stored or manipulated. These registers are often used to step through a table of values.

The SP registers hold the address of the top end of a stack. This stack is a section of memory you reserve in your program to save address and register pair values which are stored and retrieved in a last in first out organization. Each entry in the stack contains 2 bytes. The stack is mainly manipulated by the PUSH, POP, CALL and RET instructions. As entries are added to the stack, the address in the SP register is decremented by 2 with the reverse occuring as entries are removed.

You may be wondering what role assembly language plays in all of this. Rather than trying to write a program in machine language by calculating and storing in memory the series of numeric codes and address values they will require as operands, assembly language allows the user to write the same machine language program by expressing the instructions in a symbolic form.

Each possible instruction in machine language has an assembly language name. Correspondingly, operands can be expressed as names. Register I.D.s have unique names in assembly language.

When an instruction must address an area in memory in your program, the starting address of that area can be given a name and the name used in the operand to the instruction. Using this technique allows the user to place data areas anywhere in his program he wishes without knowing the actual address of that area. Likewise, names may be used to address instructions in your program which would be used to jump from one place to another within your program. This will be highlighted further when we take a look at programming examples.

Now that we have the basics, let's take a look at the actual set of instructions available to us in Z-80 machine language and how they work:

Each instruction explored will be called by its assembly language name and the characteristics covered will include the number and type of operands required by the instruction.

Before we do that, you should be familiar with how 16 bit arithmetic works and how it is applied to addresses. 16 bit arithmetic takes place in register pairs. The two registers are locked together to form a string of 16 bits which can hold a large numeric value similarly to the way a byte holds a numeric value. From left to right, each bit position containing a 1 bit has the following values which, when accumulated, make up the numeric value represented in the register pair: bit 1 is 32768, bit 2 is 16384, bit 3 is 8192, bit 4 is 4096, bit 5 is 2048, bit 6 is 1024, bit 7 is 512, bit 8 is 256, bit 9 is 128, bit 10 is 64, bit 11 is 32, bit 12 is 16, bit 13 is 8, bit 14 is 4, bit 15 is 2 and bit 16 is 1.

In order to perform 16 bit arithmetic, values must be loaded before and stored after the operation, to and from the register pair, from and to memory. For example, a 16 bit value might be the address of a table in memory you wish to increment to arrive at the table entry desired.

When not being manipulated, 16 bit values are stored in data fields in a section of memory reserved by you in your program. These two byte fields hold the values in an altered format. The first 8 bits in a 16 bit value are stored in byte 2 and the last 8 bits are stored in byte 1. This is done by store and PUSH instructions. The reverse operation is performed by load and POP instructions.

In the course of discussing how an operation code uses its operands, different types of operands will be utilized. When the register notation M is used, it means the one byte operand is in the memory location pointed to by the address in the HL register pair.

The index operand is pointed to by an address in the IX or IY registers. The operand notation may contain a displacement address value from 1 to 255 which, when present, would be added to the address in the register to arrive at the actual location of the operand byte in memory. This displacement value does not affect the contents of the IX or IY register. It is useful when scanning a table of data values in memory containing several fields of information per table entry by having the address of the start of a table entry in the IX or IY register and the relative starting position of the field within an entry, minus

one, as the displacement value. An example of the assembler operand notation for a displacement value of 1Ø would be expressed by IX + 1Ø as opposed to just IX for a displacement value of zero.

Some instructions require a flag bit condition as an operand which is used to control whether the instruction will be carried out or ignored. These condition operand codes are: 1 to test for a carry bit of 1, 2 for a carry bit of Ø (no carry), 3 for a sign bit of 1 (minus), 4 for a sign bit of Ø (positive), 5 for a zero bit of 1, 6 for a zero bit of Ø (not zero), 7 for a parity bit of 1 (parity even or signed numeric overflow), 8 for a parity bit of Ø (parity odd) and 9 for uncondition execution.

A bit number operand is used to address a single bit within a byte. The possible codes are 1 thru 8 numbering bits from left to right. In some assemblers, bits are numbered Ø thru 7 from right to left in a byte.

An immediate value operand is a numeric value ranging from Ø to 255.

An absolute address operand is a numeric address value ranging from Ø to 65535.

A label operand is a symbolically addressed location within your program using a name.

A byte operand may include either the contents of registers A, B, C, D, E, H, L or the M, immediate value or index operand values.

As you can see, the addressing possibilities used in operands can be endless in terms of combinations. The specific allowable combinations used by individual instructions will be discussed in the following review:

The ADC instruction adds its byte operand to the A register. If the carry flag was set to 1 before the instruction was executed, the A register is also incremented by 1.

The ADCX instruction adds either the BC, DE, HL or SP operand to the HL register pair. If the carry flag was set to 1 before the instruction is executed, the HL register pair is also incremented by 1.

The ADD instruction adds its byte operand to the A register. The ADDX instruction adds either the BC, DE or SP register pair in operand 2 to the HL, IX or IY register pair in operand 1. Operand 2 may also be the same as operand 1 for doubling.

The AND instruction logically ands its byte operand to the A register. In anding, the corresponding bit positions from each of the bytes involved are compared to each other one bit position at a time. If both bytes contain a 1 bit in that position, the corresponding bit position in the A register is set to 1; otherwise, it is set to Ø.

The BIT instruction examines a bit position (pointed to by the bit number in operand 1) in the byte operand in operand 2. If the bit is 1 the zero flag is set to Ø; otherwise, the zero bit flag is set to 1.

The CALL instruction is executed if the condition expressed in operand 1 exists. Operand 2 contains the absolute address or label operand locating the subroutine you wish to execute. The address of the next instruction after the call (in the PC register) is pushed onto the stack. When the subroutine executes a RET instruction, whose operand is also a condition code, the stack is popped into the PC register causing execution to continue with the instruction at the new address in the PC register.

The CCF instruction moves the carry flag bit to the half carry flag bit and the carry flag is examined. If the bit is Ø, the carry is set to 1; otherwise, the carry is set to Ø. This instruction has no operands.

The CP instruction compares the byte operand to the A register. The zero, sign, half carry, parity and carry flag bits are reset to Ø. If the comparison is equal, the zero flag is set to 1. If the A register is less than the byte operand, the carry flag is set to 1. If the A register is greater than the byte operand, a not zero flag with a no carry flag will indicate it.

The CPD instruction compares the implied M operand to the A register, setting condition flags as in the CP instruction. The HL and BC register pairs are both decremented by 1.

The CPDR instruction performs the same function as the CPD instruction except that the BC register pair is checked for a zero content after the decrement. If BC is not zero and a not equal condition resulted from the comparison, the PC register is reset to re-execute the instruction until an equal condition is found or the BC register is zero.

The CPI instruction is identical to the CPD instruction except that the HL register is incremented, not decremented, after the comparison.

The CPIR instruction is identical to the CPDR instruction except the HL register is incremented as in the CPI instruction.

The CPL instruction performs bit inversion on all the bits in the A register.

The DEC instruction subtracts 1 from the byte operand (other than an immediate value).

The DECX instruction subtracts 1 from the register pair BC, DE, HL, SP, IX or IY operand.

The DJNZ instruction decrements the B register by 1. If the result is not zero, a signed numeric value operand of between -128 to +127 would be added algebraically to the PC register after it has been adjusted by +2 for the length of the instruction machine code. This allows a jump to be made to an instruction in your program from 126 bytes before the DJNZ instruction and to 129 bytes after the DJNZ instruction. In assembler language, a nearby addressed area name (label) may be used as an operand and the assembler will compute the signed displacement value from the DJNZ instruction.

The EXS instruction exchanges the contents of the HL, IX or IY register pair operand with the contents of the top entry in the stack.

The EXD instruction exchanges the contents of the DE and HL register pairs. No operand is required.

The EXA instruction exchanges the contents of the main AF register pair with the contents of the alternate AF register pair. No operand is required.

The EXX instruction exchanges the contents of the main B, C, D, E, H and L registers with the contents of the alternate B, C, D, E, H and L registers. No operand is required.

The INC instruction is like the DEC instruction except that it adds 1 rather than subtracts 1.

The INCX instruction is like the DECX instruction except that it adds 1 rather than subtracts 1.

The JP instruction uses the condition code in operand 1 to decide if the PC register should be changed (a jump) to the named or absolute address indicated by operand 2. For an unconditional operand 1, the operand 2 address may be the M, IX or IY register.

The JR instruction performs a jump to a nearby location in your program based on the condition code in operand 1 and a signed numeric displacement value or nearby address label name similar to that used by the DJNZ instruction, as operand 2.

The LD instruction moves 1 byte areas of information from one place to another. Operand 1 is the receiveing area, operand 2 the sending area. Operand 1 may be an absolute address or address label name, M, A, B, C, D, E, H, L, index operands, an address in BC (@BC), or an address in DE (@DE). Operand 2 can be a byte operand, an absolute address or address label name, an address in BC, or an address in DE. The @BC and @DE operands (similar in operation to M) can only be used with the A register as the opposing address operand. Both operands can not be an index operand, M, an absolute address or label.

The LDX instruction moves 2 byte areas from one place to another. Operand 1, the receiving address, can be an absolute address or address label name, BC, DE, HL, IX, IY or SP; while operand 2, the sending address, can be an absolute address or address label name, an immediate address literal value (an absolute address prefixed by a #), IX, IY, SP, BC, HL or DE. Both operands can not be the same, an absolute address, label or immediate address value. The BC, DE, HL, IX or IY operand 1 possibilities can only allow the absolute address, label or immediate address value operand 2 possibilities. The SP operand 1 can have any operand 2 except BC, DE or SP.

As you can see, the LD and LDX instructions perform the load and store instruction processes.

The system hardware has 256 I/O ports, numbered Ø thru 255, used to communicate with various peripheral equipment (keyboard, cassette tape, etc.) connected to it.

The IN instruction inputs a byte into the A register from the I/O port specified in the immediate value operand.

The LDD instruction moves the byte addressed by HL to the location addressed by DE. Then BC, DE and HL are all decremented by 1.

The LDDR instruction performs the LDD function and then BC is checked for a zero content. If not zero, the instruction is repeated.

The LDI instruction works like LDD except DE and HL are incremented by 1, not decremented.

The LDIR instructions performs like LDDR except DE and HL are incremented by 1, not decremented.

The NEG instruction subtracts the value in the A register from zero and stores the result in the A register.

The NOP instruction does nothing except use up 1 machine cycle of time delay.

The OR instruction ors the corresponding bit positions of the byte operand with the A register. If either corresponding bit is 1, the A register bit is set to 1; otherwise, it is reset to Ø.

The OUT instruction outputs the byte in the A register to the I/O port specified by the immediate value operand.

The POP instruction releases the top entry of the stack into the AF, BC, DE, HL, IX or IY register pair operand and adjusts the SP register accordingly.

The PUSH instruction stores the AF, BC, DE, HL, IX or IY register pair operand onto the top of the stack and adjusts the SP register accordingly.

The RES instruction resets a bit position (specified by the bit number in operand 1) in the byte operand (other than an immediate value) specified in operand 2 to Ø.

The RL instruction shifts the bits in the byte operand (other than an immediate value) one bit position to the left and copies the carry flag bit into the rightmost bit and copies the originally leftmost bit into the carry flag bit.

The RLC instruction works like the RL instruction except the original leftmost bit is also copied into the new rightmost bit position rather than the original carry flag bit.

The RR instruction shifts the bits in the byte operand (other than an immediate value) to the right one bit position and copies the original carry flag bit into the new leftmost bit position and then copies the original rightmost bit into the carry bit.

The RRC instruction works like the RR instruction except the original rightmost bit is also copied into the new leftmost bit position rather than the original carry flag bit.

The SBC instruction subtracts the contents of the byte operand from the A

register. If the carry flag was 1 before the subtraction, the A register is also decremented by 1.

The SBCX instruction subtracts the contents of the BC, DE, HL or SP operand from the HL register pair. If the carry flag was 1 before the subtraction, the HL register is also decremented by 1.

The SCF instruction sets the carry flag to 1.

The SET instruction sets a bit position (indicated by the bit number in operand 1) in the byte operand (other than an immediate value) in operand 2, to 1.

The SLA instruction shifts the bits in the byte operand (other than an immediate value) to the left one bit position and copies the original leftmost bit into the carry flag and resets the new rightmost bit position to ∅.

The SRA instruction shifts the bits in the byte operand (other than an immediate value) to the right one bit position and copies the original rightmost bit into the carry flag and the leftmost bit position retains its original setting.

The SRL instruction works like the SRA instruction except the new leftmost bit position is reset to ∅.

The SUB instruction subtracts the contents of the byte operand from the A register.

The XOR instruction performs an exclusive or function on each of the corresponding bit positions in the byte operand with the A register bits. If only one of the two corresponding bits is 1, the corresponding A register bit is set to 1; otherwise it is reset to ∅.

These are not the only Z-80 machine instructions available. Those not covered involve more complex I/O functions, interrupt handling, BCD manipulation and some redundant instructions which the normal assembly language application program does not require. There are other assembler systems which support the full Z-80 instruction set.

Besides the Z-80 instructions, an assembly language must have some directive instructions to assign address label names and set up data area in your program. The directive instructions we have selected for our assembly language include the following:

The TAG instruction assigns the name given as an operand to the location in your program where it is found. Other assemblers assign label addresses found to the left of assembly op codes in a source program.

The ORG instruction adjusts the assembler location counter (this tells the program that converts your assembly language program into machine language for execution where to put the next converted instruction in terms of memory address. The counter is set to ∅ at the start of a program conversion and can never become negative in value. When a converted program is loaded for execution, its memory addresses will be relocated to the load address the user desires. In

this way, the counter acts as a relative address from the chosen base (load) address.) using the signed numeric operand supplied ranging in value fron -65535 to +65535. This is useful in redefining data areas in your program in terms of format.

The DS instruction reserves a block of memory for data storage starting at the location of the instruction in your program. Its numeric operand 1, ranging from 1 to 32768, specifies the number of bytes to be reserved.

The DC instruction sets up a character string literal in your program memory using the character string enclosed in quotes in operand 1.

The DB instruction sets up a byte with the 8 bits set as shown in its operand 1 containing a string of 1's and Ø's.

The DW instruction sets up a 16 bit numeric value using its numeric operand 1 value.

The DN instruction sets up a byte with the signed numeric value specified in its operand 1 value.

The DA instruction sets up a 16 bit address value field from the absolute address value or address label name specified in its operand 1.

The DM instruction sets up a byte with numeric value from Ø to 255 using the numeric operand.

The DS, DC, DB, DW, DN, DA and DM instructions can have an optional operand 2 which will assign a name to the starting address of the field being defined.

In next month's issue of THE 80 NOTEBOOK, we will cover how to use these instructions to make use of the I/O service routines (for the keyboard and cassette tape) found in the BASIC ROM memory. We will present the complete programming system to enter, edit and convert assembly language programs into machine language using cassette tape files for input and output. We will also explain how relocation works and how to pass information (fields) back and forth between BASIC and assembly language programs. A relocating loader utility program will be presented for the loading and executing of assembly language application programs and the loading and performance of assembly language subroutines by BASIC or assembly language application programs. We will also take a look at the actual machine language code generated for these assembly language instructions and how much memory it takes up. We will also take a look at some assembly language programming examples to help you get started in the interesting world of assembly language programming.

NEW PRODUCTS

TRS-80 users will be happy to learn that a Text Editor is now available for standard Level II BASIC from Southeastern Software, 512 Conway Lane, Birmingham,

AL 35210, at a cost of $40.00. A manual is optionally available for $7.50 per copy. The Southeastern Textan is a multifaceted machine language editor designed to operate with at least 16K of memory. It is a video, not line, oriented editor exclusively designed with a Basic programmer in mind. The Textan is designed such that it reads program tapes written by Level II BASIC and returns to BASIC with the program fully loaded upon completion of the edit function. Its features include thirty-two (32) command functions and twenty-six (26) reserved word keys.

The command functions allow for: top, bottom, and center of screen; end of and first of line; character, word, to end of line, and line delete; previous screen, word, and line; next screen, word, and line; search; search and replace; auto line numbering; top and end of text; line and character insert; quit insert mode; block delete; display free memory; move cursor down one line, up one line, left, and right; and tape load.

The reserved word keys will automatically enter: AND, GOSUB, CHR$, DIM, ELSE, FOR, GOTO, THEN, INPUT, RETURN, KILL, LEFT$, MID$, NEXT, OPEN, PRINT, READ, RIGHT$, STRING$, TAB(, USING, VAL, DATA, REM, LEN, and STR$ with a single keystroke.

.....

The MANAGEMENT, producer of low-cost software for TRS-80 users in the programming, small business and broadcasting areas announces a programmer utility called The FORTRANSLATOR.

The FORTRANSLATOR is designed to aid in the literal translation of TRS-80 Disk Basic Model I programs to TRS-80 FORTRAN.

This copyrighted, machine language program will run in a 32K or larger machine with at least one disk drive. A printer is desirable.

The FORTRANSLATOR converts BASIC into the structured FORTRAN "READ" / "WRITE" / "FORMAT" constructs. It also translates BASIC key-words and procedures such as IF-THEN-ELSE into the correct style. FORTRAN indentation and spacing; "C" lines; "DO" loops and other conventions are produced. Unique "GOTO" line numbers are created and subroutine "CALL" is supported. In addition, a "template" is created for FORTRAN specific items, so that the user may "plug-in" these items after translation.

The translated program is on a diskette file that is compatible with the editor in the FORTRAN package.

Depending upon the program, at least 85% of the physical work of transfering the BASIC program to FORTRAN is eliminated. Use of the FORTRANSLATOR means that a program can be created and "debugged" in BASIC and then translated to the compiled FORTRAN for fast, efficient operation.

The FORTRANSLATOR is priced at $29.95 (+tax in Texas), supplied on a Model I data diskette. Instructions for use are included with the diskette.

A version written in MicroSoft BASIC, (listings only) is available for those wishing to run on other computers, at extra cost, on special order only.

FORTRANSLATOR is available from: The MANAGEMENT, Box 111, Aledo, TX 76008.

••

QUEUE's Catalogue #3 is now available. The catalogue is a directory of educational software available for Apple, Pet, TRS-80 and Atari. Hundreds of programs from over 40 educational software publishers are grouped by computer, subject matter and grade level, and described. All the programs can be ordered directly through QUEUE. $8.95 from QUEUE, 5 Chapel Hill Drive, Fairfield, CT 06432.

••

A new monthly newsletter, MICROCOMPUTERS IN EDUCATION, will commence publication in October. The newsletter, to be published by the publishers of QUEUE, will carry reviews of educational software, new product announcements, reports on CAI in the classrooms, reviews of books and magazine articles, news of meetings, and industry news. Yearly subscriptions are $15. Microcomputers in Education, 5 Chapel Hill Drive, Fairfield, CT 06432.

••

WORKSHOPS

••

Workshops to Feature Studies on Visicalc(tm) and Z-80 Softcard(tm)

Palo Alto, CA. Avalanche Productions, Inc. today announced a new series of intensive one day workshops on microcomputer software development and licensing for the producers of consumer and small business software. Each workshop will include an in-depth case study on the development and merchandizing of selected software products.

The Workshop dates, locations, and selected case studies are:

WORKSHOP	CASE STUDY
October 17, 1980 Jack Tarr Hotel San Francisco, CA	Z-80 Softcard(tm) Vern Rayburn, Pres. Consumer Products Microsoft, Inc.
November 17, 1980 Waldorf Astoria New York, NY	Visicalc(tm) Dan Bricklin, Vice President Software Arts, Inc.
November 19, 1980 Howard Johnsons Cambridge, MA	Visicalc(tm) Dan Bricklin, Vice President Software Arts, Inc.

Richard Milewski, Editor-in-Chief of the trade newspaper InfoWorld, and a principal of an established microcomputer software house, The Software Works, Inc. will introduce and moderate the San Francisco Workshop.

Each workshop is directed toward applications programmers and analysts with experience on large and small systems who wish to understand the complex marketing, legal, and publishing issues facing the contemporary software author. The morning session will focus on techniques for identifying vertical market segments for specific product areas, and progress into an extensive discussion on strategies for packaging and marketing selected products.

Tim Barry, noted author and lecturer, will lead the early morning session in defining key software development issues for the immediate future. Mr. Barry is a co-founder of Pragmatic Designs, Inc. and is expert in both the hardware and software of microcomputers.

Later in the morning, case studies will be presented by the authors and managers directly responsible for the development and marketing of major microcomputer software products.

David Cole, Senior Computer Science Editor for CBS, Inc., will explore the issues of:

- The selection of distribution channels.
- How to evaluate a software publisher.
- Pre-contract relationships and non-disclosure agreements.
- Royalty advance requirements and schedules.
- Product maintenance issues.
- Typical contracts and agreements.

A thorough review of the copy right law and the legal mechanics of software protection will be presented by associates of Avalanche and Alan M. MacPherson, partner in the law firm of Skjerven, Morrill, Jensen, MacPherson, & Drucker. Mr. MacPherson has 15 years experience as a patent counsel to major electronic companies, and has extensive expertise in all aspects of domestic licensing of software.

Ramon Zamora, software author, educator, and management consultant will present the dollars and cents aspects of consumer software development, including sample profit and loss statements covering:

- Single projects.
- Multiple projects.
- Documentation costs and returns.
- System amortization.
- Maintenance costs.
- IRR, ROI, and NPV.

For many independent software developers, Mr. Zamora's session will prove invaluable in exposing some of the myths promulgated by dinosaur software.

The cost for each workshop is $195.00/person. Session notes and lunch are included.

For further information contact:

For further information contact:

Barbara Barnes, Avalanche Productions, Inc., 636 Waverly St., Palo Alto, CA 94301. (415) 327-0541

Late Item: Christopher P. Morgan, Editor-inChief of Byte and On Computing magazines, and Director of Byte Books will introduce and moderate the East Coast workshops.

THE 80 NOTEBOOK
5967 SULLIVAN TRAIL
NAZARETH, PA 18064

BULK RATE
U.S. POSTAGE
PAID
Stockertown, PA 18083
Permit No. 8

4/83

3/31